NUMPY ARRAY

NumPy tutorial

https://www.machinelearningplus.com/python/numpy-tutorial-part1-array-python-examples/

https://www.machinelearningplus.com/python/numpy-tutorial-python-part2/

Exercises

https://www.machinelearningplus.com/python/101-numpy-exercises-python/

-

NDARRAY & ITS AXES

np.ndarray object

- **Holds only one type of data** (dtype)
- **ndarray does not change its size**, adding cols or rows creates new object, with new id
- **can be reshaped in any way** – that does not affect item order or array size.
- element-wise math operations on ndarray are not affected by any changes in array shape. Change in shape may affect the results of linear algebra eg. dot. Product.
- Possible dimentions or array with **size = 9 are:**

(9,) -	vector, **no distinction between, rows & cols in NumPy**
(9, 1) -	array, with 9 rows and 1 column
(1, 9) -	array with 1 row and 9 columns
(3, 3) -	array with 3 rows and 3 columns

axis = 0 **operation on rows**

- results stored in new row
- Concatenate; -> **adds new rows**

```
|++++++|
|++++++|
|++++++|
|* *  * *  *  *|
```

axis = 1 **operation on columns**

- results stored in new column
- Concatenate 1; -> **adds new column**

```
|++++++| * |
|++++++| * |
|++++++| * |
```

NEW ARRAY

np.array( )	creates an array from list or tupples
	Array = np.array([[1, 2, 3], [4, 5, 6]]);
+ 1D array	np.arange(5, dtype=float))
	np.arange(1, 6, 2, dtype=int)
+ 2D array	np.array(range(6), float).reshape((2, 3))
	np.array([[1, 2, 3], [4, 5, 6]], float)
ndmin = #	Set minimum number of dimensions in an array;
	np.array([1, 2, 3], ndmin=2) # array([[1, 2, 3]])
np.arange( )	start=1, stop=10 >> 0:9:
	np.arange(start=0, stop=10, step=2, dtype="int");
aarray.copy( )	;array copy with new id()
	new_array = old_array.copy()

SPECIAL ARRAY

np.empty( )	array filled wiht random numbers not empty!
np.zeros( )	np.zeros([2,2])
np.ones( )	np.ones([2,2])
np.identity( )	np.identity(2, dtype=float) # array([1,0], [0,1])
np.eye( )	square shape array with 1 along the k-th diagonal
	np.eye(3, k=1, dtype=float);
	# [[0., 1., 0.],
	# [0., 0., 1.],
	# [0., 0., 0.]]
np.zeros_like( )	Creates a copy of an existing array with zeros
np.ones_like( )	same as above but with ones

NDARRAY CLASS

np.ndarray	class name;
type(Array)	returns class name

isinstance(Array, **np.ndarray)**; True if Array is np.ndarray.

Caution: it also returns True also for array subclasses, such as matrix

DTYPES

+ **Commonly used datatypes** # 'float', 'int', 'bool', 'str', 'object'

+ **Memory allocations:** # ‘float32’, ‘float64’, ‘int8’, ‘int16’, ‘int32’

+ “object”	to store different types of data!
dtype = “… “	parameter of np.array() , used to set up dtype in an array when it is being created, from the list of tupple.
	np.array(listName, dtype=“int”)
a.astype();	to change datatype in array:Array.astype('int').astype('str')
a.dtype	data type in array, **No brackets!**

CLASS TRANSFORMATION

np.asmatrix( )	array to numpy matrix
np.asarray( )	np matrix to np array
a.tolist( )	array to list
a.tostring( )	array to a bin str (not in human-readable '\x00\00\x00\@')
np.fromstring( )	good for saving array with big data, and read it later

SAVE & LOAD

+ in txt	txt files are bigger and slower than binary, but should work on each platform without any changes
np.savetxt( )	Array = np.array([1, 2, 3, 4]);
	np.savetxt('array.txt', Array, fmt=‘%d’) # fmt - format
np.loadtxt( )	Loaded_Array = np.loadtxt('array.txt', dtype=int)
	Array == Loaded_Array # should have only True
+ in binary	platform dependent, and may change in diff. systems
a.tofile( )	it has, sep= keyword arg.; a.tofile('test2.dat’),
np.fromfile( )	b = np.fromfile(‘a.dat', dtype=int);
+ by Python	more compact and faster to create than text files
	Platform independent
np.save( )	np.save(“a.npy”,a)
np.load()	Loaded_array = np.load(“a.npy”)
np.savez( )	output is compressed

ARRAY DIMENSIONS

INSPECT ARRAY DIMENSIONS

a.size total nr of elements in an array, **No brackets!**

np.ndim( ) number of dimensions, i.e how many axes there is in an array (only a number not the size of axes), **No brackets!**

a.shape array shape eg: 2x2, **No brackets! # (2 , 2)**

Why no brackets? Because ndim, shape, size, and dtype are **object variables**, not methods.

RESHAPE

+ why to change? (1) scikit-learn, may require 1d array of output var. to be shaped as a 2d a. ie. array with one column.

(2) Long Short-Term Memory in recurrent NN in Keras, require input as 3d array comprised of **samples (rows), timesteps (col), and features (depth).**

reshape( ) change shape, but keeps, ndim. Gives **NEW ID**

flatten( ) turns any array to 1d, new array is a copy **NEW ID**

ravel( ) any a. to 1d, new a is ref. to parent a., **NOT A COPY**

np.newaxis( ) # add new axis to an array

Array = np.array([1, 2, 3], float); Array.shape # (3,)

Array[: , np.newaxis].shape; # (3,1);

Array[np.newaxis , :].shape; # (1,3)

RESHAPE EXAMPLES

+ 2d > 1d a = np.array(list(range(1,101))).**reshape**(20,5);

b = a.**flatten**() # array turned into 1d

+ 1d > 2d col c = **b.reshape**(b.shape[0], 1)

Caution: a, b, and c have unique id's

+ 2d > 3d for Keras a = np.array(list(range(1,6)))

A_keras = **a.reshape**(a.shape[0],1,1);

A_keras.shape; (5, 1, 1)

CONCATENATION

axis = 0 **new rows**

```
|++++++|
|++++++|
|** ** * *|
|** ** * *|
```

axis = 1 **new column**

```
|++++++|****|
|++++++|****|
|++++++|****|
```

np.concatenate( ) require tuple with arrays

A = np.zeros([2, 2]); B = np.ones([2, 2])

np.concatenate((A, B), axis=0)

```
array( [   [0., 0.],
           [0., 0.],
           [1., 1.],
           [1., 1.] ] )
```

np.concatenate((A, B), axis=1)

```
array(     [[0., 0., 1., 1.],
            [0., 0., 1., 1.] ] )
```

np.r_[,] Row stack; **Caution: use square brackets [,]**

np.r_[(A, B)] # res. As in 1st example above

np.c_[,] Column stack; **Caution:** use square brackets [,], **Caution 2:** may return weird dimension for 1D objs.

np.c_[(A, B)] # res. As in 2nd example above

np.vstack( ) vertical stack, adding new rows!, useful for up to 3 dimensions, than look into concatenate, block or stack functions # a = np.ones([1,3]); b = np.zeros((1,3)); c = np.vstack((a,b)); c; array([[1., 1., 1.], [0., 0., 0.]])

np.hstack() horizontal stack, adding new elements as new columns, c = np.hstack((a,b)); c; array([[1., 1., 1., 0., 0., 0.]])

np.append() for joining two arrays, along given axis

a3 = np.append(a1,[a2], axis=0)

INDEXING & SLICING

+ SciPy Indexing https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

+ Indexing routines. ttps://docs.scipy.org/doc/numpy-1.13.0/reference/routines.indexing.html

+ INDEXING **starts at 0;** python accepts negative indexing

2D: [ROW, COLUMN] (in C# it would be a[0][0])

+ SLICING ; **[FROM : TO: STEP]; to is not included**

[1] slice with indexes

a = np.array(list(range(1,101))).reshape(10,10);

a[0:3,: -1] # first 3 rows, all columns except the last

[2] conditional selection

a = np.array([[6, 4], [5, 9]], float);

a[a >= 6] # array([6., 9.])

[3] provide indexes in an array or a list

A = np.array([2, 4, 6, 8], float);

B = np.array([0, 0, 1, 3, 2, 1], int)

A[B] # array([2., 2., 4., 8., 6., 4.])

or with list; List = [0, 0, 1, 3, 2, 1]; a[List]

take( ) A = np.arange(9).reshape([3,3]); # try by yourself.

axis=0 Select rows; eg: **A.take([1,2] ,axis=0)**

axis=1 Select columns, eg: **A.take([1,2,2,2,2] ,axis=1)**

Comments to slicing with take and int, in list or array:

- input is a list or array with row/col indexes to select
- the same index in can be selected several times
- results are stored in NEW OBJECT (new id)

a[::-1, ::-1] reverse order in all col/rows

NUMPY MATRIX

np.mat( ) creates matrix; a subclass of np.ndarray

Array = np.mat("1, 2; 3,4") # matrix([[1, 2] , [3, 4]])

Caution: isinstance(Array, np.ndarray) # True

- **Matrix has strictly 2 dimensions**, and has all methods as for array, whereas np.ndarray has unlimited number of dimensions.
- **a * b gives dot product with matrix!, a**2** - for array, give element wise square value, for matrix gives a**a dot product
- matrices behave different with **ravel method**
- **better use arrays !** np.array(np.mat("1, 2; 3,4"))

VALUES SUMMARY

STATISTICS

+ info # to extract whole array properties

np.amin(a, axis=0) / a.max() # min value in array

np.amax(a, axis=0) / a.min() # max v

+ indexes wiht

a.argmax() # index with max value inside the array

a.argmin() # min -||-

+ stats

a.mean() / np.mean()

np.mean(a, axis=0) # row/ wise

a.std()

+ products

a.sum() / np.sum()

np.cumsum(a) # cumulative sum, returns, 1d array

a.prod() /… # product of all int in an array

+ constant

np.pi / np.e # CONSTANTS in a NUMPY package!

ELEMENT-WISE OPERATIONS

*** EG:** a = np.array([1.1, 1.5, 1.9], float)

np.floor( ) # lower int. ([1., 1., 1.])

np.ceil( ) # upped int ([2., 2., 2.])

np.rint( ) # nearest integer ([1., 2., 2.])

np.abs( ) # abs value

np.sign( ) # if >0 == 1; else == 0

np.sqrt( ) / np.exp( )

np.log() / np.log10()

+ trigonometric functions # sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh, andarctanh

FINDING & SELECTING ITEMS

LOGICAL AND, OR, NOT

np.logical_and( ) fidst items meeting two coditions, returns Bool array, of the size of riginal array; np.logical_and(A<6, A>3)

np.logical_or( ) True if at least one condition was met. np.logical_or(A<6, A>3)

np.logical_not( ) True if at least none condition was met. np.logical_not(A<6, A>3)

ARRAYS COMPARISON

Arrays (A, B) must be the same dimensions, result is a NEW OBJECT with 1 dimenstion

- A > B ; eg: Truearray>Boolarray
- A == B ; eg: Truearray>Boolarray

FIND NA & INF

+ np.nan; **special object** representing missing data, it is placed instead of numbers inside the array, as the one below

+ np.inf special **object** representing infinitive data, ..

np.isnan( ) returns boolean array, np.nan==True

np.isinf( ) same as above for np.inf

np.isfinite( ) returns boolean array, with location of finite numbers **True for** all finate numbers <, ==, > from 0 **False for**, np.inf, np.nan, np.NINF, np.log(-1.), np.log(0),

FIND ZERO & TRUE

a.nonzero( ) Important - It returns a tuple that looks like a list form of input array, with ones at nonzero values and zero for zero! ITS VERY CONFUSING FUNTION.

```
a = np.array([[0, 1], [3, 0]], float);
a.nonzero() # (array([0, 1]), array([1, 0])
```

np.any(a) is any element of boolean array TRUE

np.all(a) are all elements of that array TRUE

FIND UNIQUE OR DIAGONAL VALUES

np.unique( ) **+ return_counts=True;**

if True that function returns a list with two arrays, one with unique items in an Array, second with counts for each of these items. Can be in separate arrays as below:

```
A  = np.array(["a","a","b","b","b","c","c","c","c"])
uniqs, counts = np.unique(A, return_counts=True)
#   array(['a', 'b', 'c'])  &   array([2, 3, 4]))
```

np.diagonal( ) extract diagonal value only

```
np.array([[1, 2], [3, 4]], float).diagonal();
#             [ 1.,  4. ]
```

SELECT VALUES AT RANDOM

np.random.choice( ) extract random values form array

```
np.random.choice( [ 'a',  'e',  'i' ], size = 10)
np.random.choice( [ 'a',  'e',  'i' ], size = [ 5, 5])
# can be done with probabilities and repetitions -
np.random.choice( [ 'a',  'e',  'i' ], size = 10,
                       p = [ 0.8 , 0.1 , 0.1 ] )
```

SORT

a.sort() / sorted(a) beast read onlkine instructions before using

a[::-1, ::-1] reverse order in all col/rows

MODIFY VALUES IN ARRAY

A.fill( ) Fill in array with a single value at each index;

PUT, COPYTO & WHERE

a.put( ) Change elements in array based on condition and input values

```
A = np.array([0, 1, 2, 3, 4, 5], float);
B = np.array([9, 8, 7], float);
A.put( [0, 3], B);
#          array([ 9.,  1.,  2.,  8.,  4.,  5.])
```

Comments: in put we place indexes in "A" where we place items form "B" . new items are takes in order of appearance in B. Thus we cvan only control when and which item is placed in A. eg. the value 7 from the source array b is not used, since only two indices [0, 3] were specified. The source array B will be broadcasted if necessary

np.copyto( ) **IMPORTANT: it doesn't create new array**
Copies values from one array to another, broadcasting if necessary. np.copyto(dst, src, where=None)

eg: copy zeros from B into A, where C==True

```
A = np.ones([3,3])
B = np.zeros([3,3])
C = np.reshape(np.arange(1,10)>4, [3,3])
np.copyto(A, B, C)
#          array([    [1., 1., 1.],
#                     [1., 0., 0.],
#                      [0., 0., 0.]])
```

np.where( ) Works, like, if else for array. Merges, two arrays, using True False information in boolean array. Requires three arrays of the same size Results stored as **NEW OBJECT.**

.1. Boolarray; all items are True/False)

.2. Truearray,array with values used as results, if True in 1.

.3. Falsearray; a. with values used as results, if False in 1.

```
Array = np.arange(1,10); Boolarray = Array>4
Truearray = np.ones(9,); Falsearray = np.zeros(9,)
np.where(Boolarray,Truearray, Falsearray).reshape(3,3)
#          array([    [0., 0., 0.],
                      [0., 1., 1.],
                      [1., 1., 1.]]) # new array
```

CLIP & MASK

a.clip() To filter the data with a thresholds

i) **upper and lower thresholds**

ii) Replaces, numbers below/above thresholds with it.

iii) Returns a view; copy the result to new array

```
A = np.arange(9).reshape(3,3)
B = A.clip(2,4).copy(); B
          array([    [2, 2, 2],
                     [3, 4, 4],
                     [4, 4, 4] ])
```

np.putmask( ) Changes elements of an array based on conditional and input values. putmask(a, mask, values).

Two ways of making a mask:

[1] using one array, to set up the condition, and a mask

```
A = np.arange(1, 10).reshape([3, 3])
np.putmask(A, A>4, A*100); A
#          array([    [  1,   2,   3],
#                     [  4, 500, 600],
#                     [700, 800, 900]])
```

[2] using 1D array with exact or, smaller nr of el's that should be placed instead of masked elements.

```
A = np.arange(1, 10).reshape([3, 3])
B = np.arange(1, np.sum(A>4))
np.putmask(A, A>4, B); A
#          array([    [1, 2, 3],
#                     [4, 1, 2],
#                     [3, 4, 1]])
```

ITEARTTION OVER ARRAY

+ info https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.nditer.html#arrays-nditer

+ simple for loop: will print each row, wiht no access to it.

+ iterate change and value

np.nditer( ) allows using for: loops on array items(i).

.... **op_flags=['readwrite'];** otherwise we can only read

.... **i[...] =;** assignment, to place new item in the same cell as original item, like inplace = True, in Pandas

.... Example:

```
Array = np.arange(1, 10, 1)
for i in np.nditer(Array,  op_flags = ['readwrite'] ):
          i[...]  = 20.    # all values replace with 20
          i[...]  = i +13 # all values + 13
#         array( [ 33, 33, 33, 33, 33 ] )
#         returns 1d array
```

np.ndenumerate( ) Return pairs of array coordinates – in brackets (,) and values**. Caution:** if you don't use two arguments (index, item, like below), you will get a list with both inside eg. ((0,0), 1).

```
Array = np.arange(1,5,1).reshape([2,2])
for index, item in np.ndenumerate(Array):
print(index, item)
#         (0, 0) 1
#         (0, 1) 2
#         (1, 0) 3
#         (1, 1) 4
```

np.ndindex( ) Creates iterator with indices of array with given shape:

```
for index in np.ndindex(2, 2):
          print(index, end="; ")
#         (0, 0); (0, 1); (1, 0); (1, 1);
```

ELEMENT-WISE OPERATIONS

All arrays must have the same dimensions

a + b	**a - b**
a * b	**a / b**
a%b	**a**b**

SEQUENCES & RAND. NUMBERS

WITH UNIFORM NUMBERS

np.zeros( ) np.zeros([2,2])

np.one( ) np.ones([2,2])

np.eye( ) square shape matrices with ones along the k-th diagonal;

WITH NUMBERS FROM A GIVEN RANGE

np.arange( ) # :)

np.linspace( ) generate nr's on equal intervals in a given range!

```
np.linspace(1,100, 5, dtype="int")
#           array([  1,  25,  50,  75, 100])
```

np.logspace( ). np.logspace(start=1, stop=10, num=5, base=2, dtype="int") # [2   9 45 215 1024]

np.set_printoptions(precision=i)

# i - number of decimal in float that is being displayed

BASED ON REPTITION

np.tile(list, n)

```
np.tile([1,2,3], 2)
#           [1,2,3,1,2,3];
np.tile([[1, 2, 3],[1, 2, 3]], 2)
#            [[1,2,3,1,2,3],[1,2,3,1,2,3]]
```

np.repeat(list, n)

```
np.repeat([1,2,3], 2)
#           [1, 1, 2, 2, 3, 3];
np.repeat([[1, 2, 3],[1, 2, 3]], 2)
#           [1 1 2 2 3 3 1 1 2 2 3 3]
```

WITH RANDOM NUMBERS

+ numpy.random.rand vs numpy.random.random:

Both functions generate samples from uniform distribution [0, 1). The only difference is in how the arguments are handled. With **numpy.random.rand** , the length of each dimension of the output array is a separate argument. With **numpy.random.random** , the shape argument is a single tuple.

np.random.rand( ) rand. nr's between [0,1):

```
np.random.rand( 2, 2 )
#           array([     [0.75140886, 0.68415163],
#                       [0.25143795, 0.18884039]])
```

np.random.random( ) rand. nr's between [0,1)

```
np.random.random()# gives one number
np.random.random( size = [2, 2] )
#           array([     [0.75140886, 0.68415163],
#                       [0.25143795, 0.18884039]])
```

np.random.randint( ) rand. integers: (from, to, a. size), to is excluded

```
np.random.randint(1,101,size=[3,3])
#           array([     [71, 20, 49],
#                       [21, 90, 37],
#                       [ 3, 41, 40]])
```

WITH NUMBERS FROM DISTRIBUTION

np.random.randn(); sample from norm distr. with mean=0, sd=1

Importnat: you can't change mean and sd!

```
np.random.randn(2,2) # dims of an array
#           array([     [ 1.36914215, -0.03394335],
#                       [-1.08814754, -0.34475432]])
```

np.random.normal() sample from norm distr. with custom mean & sd

- loc mean
- scale sd
- size dimensions of arr with res. , given in [,]

```
np.random.normal(loc=10,scale=5, size=[2,2])
#           array([     [16.77922167, 11.34211407],
                        [13.97457648,  1.19622091]])
```

negative_binomial(n, p[, size]) as below;

binomial(n, p[, size]) Draw samples from a binomial distribution.

```
rn.binomial(n=4 , p=0.8, size=[1,4])
# array [ [ 2 2 4 4 ] ]
# that is nrs of successes in 4 tosses, p=0.8
```

+ other distribution available https://docs.scipy.org/doc/numpy/reference/routines.random.html

- **negative_binomial(n, p[, size])**
- **poisson([lam, size])**
- **uniform([low, high, size])**
- **gamma(shape[, scale, size])**
- **beta(a, b[, size])**
- **chisquare(df[, size])**
- **dirichlet(alpha[, size])**

PERMUTATIONS

shuffle(x) Modify a sequence in-place by shuffling its contents.

permutation(x) Randomly permute a seq., or return a permuted range.

WITH NUMBERS FROM DISTRIBUTION

np.random.seed() set seed to reapeat all random combinations!

np.random.get_state() Return a tuple representing the internal state of the generator.

np.random.set_state(state) Set the internal state of the generator from a tuple.

np.random.RandomState(); creates: Class **mtrand.RandomState;** used to avoid seeing seed:

```
np.random.randint(0, 10, size = [ 2, 2 ] )
#           then:
rn.rand(1,2)
#           array( [ [8, 1]]);
rn.binomial(n=4 , p=0.8, size=[1,4])
#           array [ [ 2 2 4 4 ] ]
```